REMARKS

Claims 1, 14, 24, 29, and 39-41 are amended. Claims 1-41 remain in the application for consideration. In view of the following amendments and remarks, Applicant respectfully solicits allowance of the application and furtherance onto issuance.

Drawing Objections

Figs. 4-6 have been objected to as having unacceptable left margins. Applicant submits herewith replacement drawings thus overcoming the rejection.

§ 102 Rejections

Claims 1-23 and 29-41 stand rejected under 35 U.S.C. § 102(b) as being anticipated by U.S. Patent No. 5,379,432 to Orton. Further, claims 24-28 stand rejected under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent No. 6,144,377 to Oppermann.

Before discussing the claims in detail and to assist in appreciating the differences between the claimed subject matter and Orton's disclosure, the following discussion of Orton is provided.

The Orton Reference

Orton is directed to an object-oriented interface for a procedural operating system. In fact, in laying out the problem to which Orton's disclosure is directed to solving, Orton discusses conventional operating systems. For example, in column 3, lines 10-46, Orton states the following:

> Most conventional operating systems are ***procedure-oriented*** and include ***native procedural interfaces***. Consequently, the services provided by these operating systems can only be accessed by using the procedures defined by their respective procedural interfaces. If a program needs to access a service provided by one of these procedural operating systems, then the program must include a statement to make the appropriate operating system procedure call. This is the case, whether the software program is object-oriented, procedure-oriented, rule based, etc. Thus, conventional operating systems provide ***procedure-oriented environments*** in which to develop and execute software. Some of the advantages of OOT are lost when an object-oriented program is developed and executed in a procedure-oriented environment. This is true, since all accesses to the procedural operating system must be implemented using procedure calls defined by the operating system's native procedural interface. Consequently, some of the modularity, maintainability, and reusability advantages associated with object-oriented programs are lost since it is not possible to utilize classes, objects, and other OOT features to their fullest extent possible.
>
> One solution to this problem is to develop object-oriented operating systems having native object-oriented interfaces. While this ultimately may be the best solution, it currently is not a practical solution since the resources required to modify all of the major, procedural operating systems would be enormous. Also, such a modification of these procedural operating systems would render useless thousands of procedure-oriented software programs. Therefore, ***what is needed is a mechanism for enabling an object-oriented application to interact in an object-oriented manner with a procedural operating system having a native procedural interface.***

Thus, the gist of Orton's invention is a mechanism for enabling an object-oriented application to interact in an ***object-oriented manner*** with a ***procedural operating system having a native procedural interface***.

Before describing his specific solution, Orton characterizes his invention as an "object-oriented wrapper since it operates to wrap a procedural operating system with an object-oriented software layer such that an object-oriented application can access the operating system in a object-oriented manner." See, e.g. column 5, lines 5-25. Further on, Orton sets forth a ***specific requirement*** that is

necessary for operation of his invention. Specifically, in column 5, lines 64-68, Orton states, "[f]or purposes of the present invention, the only requirement is that the operating system 114 be a procedural operating system having a native procedural interface."

In describing his specific invention, Orton describes wrapper 128 in Fig 1 as follows (see column 6, lines 41-57):

> The wrapper 128 is directed to a mechanism for providing the operating system 114 with an object-oriented interface. The wrapper 128 enables the object-oriented applications 130A, 130B to directly access in an object-oriented manner the ***procedural operating system*** 114 during run-time execution of the applications 130A, 130B on the computer platform 102. The wrapper 129 is conceptually similar to the wrapper 128. The wrapper 129 provides an IBM PS/2 interface for the operating system 114, such that the application 132 can directly access in a PS/2 manner the procedural operating system 114 (assuming that the application 132 is adapted to operate in the IBM PS/2 environment). The discussion of the present invention shall be limited herein to the wrapper 128, which provides an object-oriented interface to a procedural operating system having a native procedural interface.

Thus, from the onset, two explicit requirements in Orton are that (1) the application (which utilizes Orton's wrapper) be an object-oriented application that makes object-oriented calls, and (2) the operating system be a procedural operating system that receives procedure calls.

Applicant's Disclosure

There are several aspects discussed in Applicant's disclosure, which is directed to overcoming problems associated with prior operating systems that

utilize large numbers of callable functions or, in Orton's words, procedural operating systems. As set forth in Applicant's background section:

> Operating systems typically include large numbers of callable functions that are structured to support operation on a single host machine. When an application executes on the single host machine, it interacts with the operating system by making one or more calls to the operating system's functions.
>
> Although this method of communicating with an operating system has been adequate, it has certain shortcomings. One such shortcoming relates to the increasing use of distributed computing, in which different computers are programmed to work in concert on a particular task. Specifically, operating system function libraries can severely limit the ability to perform distributed computing.
>
> Fig. 1 illustrates the use of functions in prior art operating systems. Fig. 1 shows a system 20 that includes an operating system 22 and an application 24 executing in conjunction with the operating system 22. In operation, the application 24 makes calls directly into the operating system when, for example, it wants to create or use an operating system resource. As an example, if an application wants to create a file, it might call a "CreateFile" function at 26 to create the file. Responsive to this call, the operating system returns a "handle" 28. A "handle" is an arbitrary identifier, coined by the operating system to identify a resource that is controlled by the operating system. In this example, the application uses handle 28 to identify the newly created file resource any time it makes subsequent calls to the operating system to manipulate the file resource. For example, if the application wants to read the file associated with handle 28, it uses the handle when it makes a "ReadFile" call, e.g. "ReadFile (handle)". Similarly, if the application wants to write to the file resource it uses handle 28 when it makes a "WriteFile" call, e.g. "WriteFile (handle)".
>
> One problem associated with using a handle as specified above is that the particular handle that is returned to an application by the operating system is only valid for the process in which it is being used. That is, without special processing the handle has no meaning outside of its current process, e.g. in another process on a common or different machine. Hence, the handle cannot be used across process or machine boundaries. This makes computing in a distributed computing system impossible because, by definition, distributed computing takes place across process and machine

boundaries. Thus, current operating systems lack the ability to name and manipulate operating system resources on remote machines.

Another problem with traditional operating system function libraries is that individual functions cannot generally be modified without jeopardizing the operation of older versions of applications that might depend on the particular characteristics of the individual functions. Thus, when an operating system is upgraded it typically maintains all of the older functions so that older applications can still use the operating system.

In prior art operating systems, a function library essentially defines a protocol for communicating with an operating system. When operating systems are upgraded, the list of functions that it provides typically changes. Specifically, functions can be added, removed, or changed. This changes the protocol that is used between an application and an operating system. As soon as the protocol is changed, the chances that an application will attempt to use a protocol that is not understood by the operating system, and vice versa increase.

Prior art operating systems attempt to deal with new versions of operating systems by using so-called version numbers. Version numbers are assigned to each operating system. Applications can make specific calls to the operating system to ascertain the version number of the operating system that is presently in use. For example, when queried by an application, Windows NT 4 returns a "4" and Windows NT 5 returns a "5". The application must then know what specific protocol to use when communicating with the operating system. In addition, in order for an operating system to know what operating system version the application was designed for, a value is included in the application's binary. The operating system can then attempt to accommodate the application's protocol.

The version number system has a couple of problems that can adversely affect functionality. Specifically, a typical operating system may have thousands of functions that can be called by an application. For example, Win32, a Microsoft operating system application programming interface, has some 8000 functions. The version number that is assigned to an operating system then, by definition, represents all of the possibly thousands of functions that an operating system supports. This level of description is undesirable because it does not provide an adequate degree of resolution. Additionally, some operating systems can return the same version number. Thus, if the operating systems are different (which they usually are), then returning the same version number can lead to operating

errors. What is needed is the ability to identify different versions of operating systems at a level that is lower than the operating system level itself. Ideally, this level should be at or near the function level so that a change in just one or a few functions does not trigger a new version number for the entire operating system.

Thus, as noted in Applicant's "Background" section, various embodiments are directed to overcoming and solving problems associated with prior art operating systems, and particularly procedural operating systems such as the one that is specifically required by Orton.

In Applicant's "Overview" section on page 6, a high level discussion of aspects of Applicant's disclosure is provided. This gives somewhat of a high level view of some of the described embodiments, and can provide a basis from which some fundamental differences between Orton and some of the individual claimed embodiments can be appreciated. As specifically set forth starting on page 6:

> In accordance with one embodiment, one or more of an operating system's resources are defined as objects that can be identified and manipulated by an application through the use of object-oriented techniques. Generally, a resource is something that might have been represented in the prior art as a particular handle "type." Examples of resources include files, windows, menus and the like.
>
> Preferably, all of the operating system's resources are defined in this way. Doing so provides flexibility for distributed computing and legacy support as will become apparent below. By defining the operating system resources as objects, without reference to process-specific "handles," the objects can be instantiated anywhere in a distributed system. This permits responsibility for different resources to be split up across process and machine boundaries. Additionally, the objects that define the various operating system resources can be identified in such a way that applications have no trouble calling the appropriate objects when they are running. This applies to whether the applications know they are running in connection with operating system resource objects or not. If applications are unaware that they are running in connection with operating system resource objects, e.g. legacy applications, a mechanism is provided for translating calls for

the functions into object calls that are understood by the operating system resources objects.

In addition, factorization schemes are provided that enable an operating system's functions to be re-organized and redefined into a plurality of object interfaces that have methods corresponding to the functions. In preferred embodiments, the interfaces are organized to leverage advantages of interface aggregation and inheritance.

The Claims Rejected Over Orton

Claim 1 recites a method of factoring operating system functions. In making out the rejection of this claim, the Office states that Orton teaches the subject matter of this claim. Applicant respectfully disagrees and traverses the Office's rejection.

In accordance with the claimed method, criteria are defined that governs how functions of an operating system are to be factored into one or more groups. Functions are factored into one or more groups based upon the criteria and groups of functions are associated with programming objects that have data and methods, wherein the methods correspond to the operating system functions. This claim has been amended to clarify that the association of the groups of functions with programming objects is effective to provide an object oriented operating system. Support for this limitation can be found throughout Applicant's specification and, in particular, in Fig. 3 and the related discussion.

Nowhere does Orton disclose or suggest a method that is effective to provide an object oriented operating system. Rather, Orton specifically teaches directly away from any such subject matter by specifically stating that a requirement of his invention is that the operating system be a procedural operating system. Accordingly, for at least this reason, claim 1 is allowable.

Claims 2-13 depend either directly or indirectly from claim 1 and are allowable as depending from an allowable base claim. These claims are also allowable for their own recited features which, in combination with those recited in claim 1, are neither shown nor suggested in the references of record, either singly or in combination with one another.

Further, with respect to claim 7 which recites instantiating a plurality of programming objects across a machine boundary, the Office notes that Orton is silent in this regard. The Office goes on to state, in a conclusory fashion, that it would have been obvious to make this modification to provide objects that communicate across machine boundaries. Applicant strongly disagrees with and traverses the Office's conclusory and unsubstantiated argument. Claim 7, when taken in combination with claim 1, patentably distinguishes over Orton.

Claim 14 recites a method of factoring operating system functions. In making out the rejection of this claim, the Office states that Orton teaches the subject matter of this claim. Applicant respectfully disagrees and traverses the Office's rejection.

In accordance with the recited method, a plurality of operating system functions that are used in connection with operating system resources are factored into first groups based upon first criteria. The first groups are factored into individual sub-groups based upon second criteria. Each sub-group is assigned to its own programming object interface, wherein a programming object interface represents a particular object's implementation of its collective methods. This claim has been clarified to recite that the assigning of each sub-group is effective to provide an object-oriented operating system. As Orton neither discloses nor suggests any such subject matter, this claim is allowable.

Claims 15-23 depend either directly or indirectly from claim 14 and are allowable as depending from an allowable base claim. These claims are also allowable for their own recited features which, in combination with those recited in claim 14, are neither shown nor suggested in the references of record, either singly or in combination with one another.

Claim 29 recites an operating system application program interface embodied on a computer-readable medium. The recited interface comprises a plurality of object interfaces. The claim has been amended to clarified that each object interface is associated with an object that includes one or more methods that are associated with and can call functions of an operating system that does not comprise the object interfaces. In addition, the claim has been amended to clarify that the individual objects are configured to be instantiated in process, locally, or remotely. Nowhere does Orton disclose or even suggest any such subject matter. Accordingly, this claim is allowable.

Claims 30-35 depend either directly or indirectly from claim 29 and are allowable as depending from an allowable base claim. These claims are also allowable for their own recited features which, in combination with those recited in claim 29, are neither shown nor suggested in the references of record, either singly or in combination with one another.

Claim 36 recites an ***operating system***. In making out the rejection of this claim, the Office states that Orton teaches the subject matter of this claim. Applicant respectfully disagrees and traverses the Office's rejection.

The recited operating system comprises a plurality of programming objects having interfaces, wherein the programming objects represent operating system resources, and wherein the interfaces define methods that are organized in

accordance with whether they create an operating system resource or not. Further, the programming objects are recited to be configured to be called either directly or indirectly by an application. Additionally, the methods are recited to be configured to call operating system functions responsive to being called directly or indirectly by an application.

The Office argues that Orton's wrapper class library meets the recited programming objects. Applicant strongly disagrees. Orton's wrapper library class is a ***wrapper*** for a ***procedural operating system***. Orton's wrapper 128 is best viewed in Fig. 1 where it is apparent that the wrapper is, necessarily, external to and does not comprise part of the operating system 114. Rather, the wrapper wraps an otherwise procedural operating system. The recited subject matter, on the other hand, recites an operating system that comprises a plurality of programming objects having interfaces. An illustrative diagram of one exemplary operating system appears in Applicant's Fig. 3. This, together with Orton's specific requirement that the operating system with which its invention operates be a procedural operating system, clearly indicate that claim 36 recites patentable subject matter. Accordingly, for at least this reason, this claim is allowable.

Claims 37-40 depend either directly or indirectly from claim 36 and are allowable as depending from an allowable base claim. These claims are also allowable for their own recited features which, in combination with those recited in claim 36, are neither shown nor suggested in the references of record, either singly or in combination with one another. In addition, claims 39 and 40 have been cosmetically amended to remove the term "application program interface" from the preamble of the claims so as to bring the claims into conformity with the subject matter recited in claim 36.

Claim 41 recites a method of converting an operating system from a non-object-oriented format to an object oriented format, wherein the operating system includes a plurality of operating system functions that are callable to create or use operating system resources. In accordance with the recited method, a plurality of programming object interfaces are defined that define methods that correspond to the operating system functions. This claim has been amended to clarify that the programming objects that support the interfaces are callable either directly by an application that makes object-oriented calls, or indirectly by an application that makes function calls. The claim has been further amended to clarify that a programming object interface is called either directly via an object-oriented call, or indirectly via an indirection that transforms a function call into an object-oriented call. The claim further recites that responsive to said call, an operating system function is called with a method of the programming object that supports programming object interface.

In making out the rejection of this claim, the Office argues that Orton's wrapper meets the recited programming object interface. Applicant disagrees, particularly in view of the claim amendment. Specifically, Orton's wrapper receives only object-oriented calls from an object-oriented application. Orton's wrapper does not receive indirect calls via an indirection that transforms a function call into an object-oriented call. To this extent, Orton teaches directly away from the recited subject matter.

Accordingly, this claim is allowable.

The Claims Rejected Over Oppermann

Claims 24-28 are rejected under §102(e) over Oppermann. Oppermann is directed to an architecture that enables an accessibility aid to directly access and manipulate user interface elements of an application program programmatically.

Claim 24 recites a method of factoring operating system functions. In accordance with the recited subject matter, a plurality of operating system functions are factored into interface groups based upon the resources with which a function is associated. The interface groups are factored into interface sub-groups ***based upon each function's use of a handle*** that represents a resource. The interface sub-groups are organized so that at least one of the interface sub-groups inherits from at least one other of the interface sub-groups.

In making out the rejection, the Office contends that Oppermann meets the subject matter of this claim. Specifically, the Office argues that Oppermann teaches a plurality of operating system functions, a handle, a resource, and organizing the interface sub-groups so that at least one of the interface sub-groups inherits from at least one other of the interface sub-groups. Applicant respectfully disagrees with the Office's interpretation of this reference and traverses the rejection.

Specifically, claim 24 recites a step in which the interface groups are factored into interface sub-groups ***based upon each function's use of a handle*** that represents a resource. Nowhere does Oppermann teach or even suggest this feature. The Office simply states that Oppermann teaches a handle and apparently contends that the mere presence of a handle anticipates this claim feature. Applicant respectfully but strongly disagrees. Oppermann describes its use of a handle starting in column 9, line 14:

FIG. 7 depicts a flowchart of the steps performed by the AccessibleObjectFromPoint function provided by the access component. The first step performed by the AccessibleObjectFromPoint function is to obtain the window handle and the window class for the user interface element at the indicated location on the video display (step 702). The AccessibleObjectFromPoint function performs this functionality by invoking the well-known WindowFromPoint function of the operating system to obtain the window handle and the window class. After obtaining this information, the AccessibleObjectFromPoint function instantiates an IAccessible interface for the particular type (or class) of user interface element (step 704). As is further discussed below, the access component maintains an interface data structure for each of the various types of windows that it supports (e.g., dialog interface class, edit interface class, and button interface class), and in this step, the access component creates an instance of the interface by invoking the well-known "new" operator of the C++ programming language. In addition, the AccessibleObjectFromPoint function initializes the function members and the properties on the interface. For example, one of the properties on the IAccessible interface is the name property, and in this step, the AccessibleObjectFromPoint function stores the name of the user interface element into this property. Additionally, the window handle obtained in step 702 ("the current window handle") is stored as a data member of the IAccessible interface, so that it may be later used by the function members on the interface. After instantiating an JAccessible interface, the AccessibleObjectFromPoint function returns a pointer to this interface to the client (step 706). The processing of both the AccessibleObjectFromWindow and the AccessibleObjectFromEvent functions is similar to that described in FIG. 7, except that in step 702, these functions need only obtain the window class because the functions receive the current window handle as a parameter.

Thus, what Oppermann is describing is simply the use of a handle. Oppermann neither discloses nor suggests factoring interface groups into interface sub-groups ***based upon each function's use of a handle*** that represents a resource. Accordingly, for at least this reason, this claim is allowable.

Claims 25-28 depend either directly or indirectly from claim 24 and are allowable as depending from an allowable base claim. These claims are also allowable for their own recited features which, in combination with those recited in claim 24, are neither shown nor suggested in the references of record, either singly or in combination with one another.

Conclusion

All of the claims are in condition for allowance. Accordingly, Applicant requests a Notice of Allowability be issued forthwith. If the Office's next anticipated action is to be anything other than issuance of a Notice of Allowability, Applicant respectfully requests a telephone call for the purpose of scheduling an interview.

Amended Claims with Markups to Shows Amendments

1. (Amended) A method of factoring operating system functions comprising:

defining criteria that governs how functions of an operating system are to be factored into one or more groups;

factoring the functions into one or more groups based upon the criteria; and

associating groups of functions with programming objects that have data and methods, wherein the methods correspond to the operating system functions effective to provide an object oriented operating system.

14. (Amended) A method of factoring operating system functions comprising:

factoring a plurality of operating system functions that are used in connection with operating system resources into first groups based upon first criteria;

factoring the first groups into individual sub-groups based upon second criteria; and

assigning each sub-group to its own programming object interface, wherein a programming object interface represents a particular object's implementation of its collective methods effective to provide an object-oriented operating system.

24. (Amended) A method of factoring operating system functions comprising:

factoring a plurality of operating system functions into interface groups based upon the resources with which a function is associated;

factoring the interface groups into interface sub-groups based upon each function's use of a handle that represents a resource; and

organizing the interface sub-groups so that at least one of the interface sub-groups inherits from at least one other of the interface sub-groups.

29. (Amended) An operating system application program interface embodied on a computer-readable medium comprising a plurality of object interfaces, wherein each object interface is associated with an object that includes one or more methods that are associated with and can call functions of an operating system that does not comprise the object interfaces, individual objects being configured to be instantiated in process, locally, or remotely.

39. (Amended) The operating system [application program interface] of claim 36, wherein at least some of the objects are disposed across at least one process boundary and at least one machine boundary.

40. (Amended) The operating system [application program interface] of claim 36, wherein the objects comprise COM objects.

41. (Amended) A method of converting an operating system from a non-object-oriented format to an object oriented format, wherein the operating system includes a plurality of operating system functions that are callable to create or use operating system resources, the method comprising:

defining a plurality of programming object interfaces that define methods that correspond to the operating system functions, wherein programming objects that support the interfaces are callable either directly by an application that makes object-oriented calls, or indirectly by an application that makes function calls;

calling a programming object interface either directly via an object-oriented call, or indirectly via an indirection that transforms a function call into an object-oriented call; and

responsive to said calling, calling an operating system function with a method of the programming object that supports said programming object interface.

Respectfully Submitted,

Dated: 1/2/03

By: _____

Lance R. Sadler
Reg. No. 38,605
(509) 324-9256